

AD-A169 223

Productivity Engineering in the UNIX¹ Environment

An Ada² Package for Dimensional Analysis

Technical Report

S. L. Graham
Principal Investigator

(415) 642-2059

"The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government."

Contract No. N00039-84-C-0089

August 7, 1984 - August 6, 1987

Arpa Order No. 4871



¹UNIX is a trademark of AT&T Bell Laboratories

²Ada is a trademark of the U.S. Government (Ada Joint Program Office).

This document has been approved
for public release and sale; its
distribution is unlimited.

86 4 3 008



Accession For	
NTIS	GRA&I
DTIC	TAB
Unannounced	
Initiation	
<i>Little Or file</i>	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A1	

An AdaTM Package for Dimensional Analysis*

Paul N. Hilfinger

University of California, Computer Science Division
Berkeley, California 94720

Abstract

This paper illustrates the use of Ada's abstraction facilities—notably operator overloading and type parameterization—to define an oft-requested feature: a way to attribute units of measure to variables and values. The definition given allows the programmer to specify units of measure for variables, constants, and parameters; checks uses of these entities for dimensional consistency; allows arithmetic between them, where legal; and provides scale conversions between commensurate units. It is not constrained to a particular system of measurement (such as the metric or English systems.) While the definition is in standard Ada and requires nothing special of the compiler, certain reasonable design choices in the compiler, discussed here at some length, can make its implementation particularly efficient.

1. Introduction

Dimensional analysis is a consistency check common in scientific and engineering computations. It consists of verifying that uses of numerical quantities conform to their declared units of measure—for example, that measures of length are not added to measures of time and that quotients of distance and time are treated as velocities. In many ways, dimensional analysis resembles strong typing, as found in Ada and other modern programming languages. Like strong typing, it usually can be performed statically—on the program text before program execution. Also as with strong typing, the information needed to perform the checks consists of a property to be attributed to each named quantity (variable, named constant, or parameter.)

In Ada, one could use strong typing together with operator overloading to get the desired checks. For example, consider the prospective package interface shown in Fig. 1. I've defined functions (mostly within the ellipses) for each of the legal operations between various kinds of quantities. There are no operations allowing one to add, for example, TIMES to LENGTHs, and therefore any attempt to do so will cause a type error (or possibly an "undefined operator" error, depending on the compiler.) This, of course, is just what we want. On the other hand, certain of the functions generated by the derived type declarations are dimensionally invalid. For example, the declaration of LENGTH generates an implicit declaration of a function that multiplies LENGTHs to give a LENGTH.

* The work reported here was sponsored by Defense Advance Research Projects Agency (DoD) Arpa Order No. 4871 Monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089. Ada is a registered trademark of the United States Government.

```

package STRAW_UNIT_PACKAGE is
  type LENGTH is new FLOAT;
  type TIME is new FLOAT;
  type MASS is new FLOAT;
  type VELOCITY is new FLOAT;
  type AREA is new FLOAT;
  ...
  -- Cross-product operators
  function "*" (L, R : LENGTH) return AREA;
  function "*" (L : FLOAT; R : LENGTH) return LENGTH;
  function "/" (L, R : LENGTH) return FLOAT;
  function "/" (L : AREA; R : LENGTH) return LENGTH;
  ...
  -- Illegal functions: always raise CONSTRAINT_ERROR exception.
  function "*" (L, R : LENGTH) return LENGTH;
  function "/" (L, R : LENGTH) return LENGTH;
  ...
end STRAW_UNIT_PACKAGE;

```

Figure 1. Sketch of a possible units-of-measure package.

For these functions, therefore, there are overriding declarations of functions whose bodies simply raise an exception.

The problems with this solution are obvious. The number of functions that must be declared is large and necessarily incomplete, since there are infinitely many possible units that one can compound from arithmetic combinations of LENGTHs, TIMEs, and MASSes. The package is not easy to expand. Finally, although perhaps the point is only of interest to academics, this solution is rather inelegant. One wonders whether there might be a better solution—one that makes better use of Ada's abstraction facilities.

The answer is a qualified yes. As I'll show in the next section, one can write an Ada package that solves the problem (almost) in general, allowing its user to define a basis set of units (such as LENGTH, MASS, and TIME), to get automatically *all* computed units (such as area or velocity), and to have the desired consistency checking performed automatically. The catch is that for most efficient operation, the package makes some demands upon the compiler. In section 3, we shall see how reasonable these demands are. Even given a non-optimizing compiler, however, the package will function correctly.

2. The Ada package

The basic idea for the package in this section is, as far as I know, due originally to John Nestor, who was the chief designer of the RED language for Intermetrics, Inc. In 1979, he informally gave a definition of a dimensional analysis package in RED, which I later modified for Ada [2]. Here, I'll present an improved version of the latter package. The definition has benefited greatly from examining another Ada dimensional analysis package

written by Narain Gehani [1]. Readers that haven't encountered the programming issues before may find Karr and Loveman's discussion [3] helpful.

In the rest of this section, I'll define a package called UNITS, which we'll consider in three parts. We'll start from the user's perspective (i.e., from that of the programmer who writes "with UNITS") and consider what he must write to get the desired effect. Next, we'll look at the necessary package specification, and then at the corresponding body.

2.1. The User's View

The user has several functional requirements of the prospective UNITS package.

1. It must be possible to declare each variable, constant, and parameter to have a particular unit of measure.
2. It must be possible to perform the ordinary arithmetic operations between quantities having declared units of measure.
3. There must be dimensionless (scalar) quantities, which can be multiplied and divided by quantities having declared units of measure.
4. It must be possible to recover the numeric values of quantities with declared units (in particular, for output.)
5. It must be possible to have literals with units (so that one can insert a constant such as 2.54 cm. into a program, for example.)
6. Violations of dimensional consistency—such as assignments of quantities having the wrong units for the receiving variable, and additions of quantities with incommensurable units—must be detected, preferably at compilation time.
7. It should be possible to assign names to new units of measure, so that they may be referenced symbolically.
8. There should be some provision for handling conversion of commensurable units.*

Consider, for example, the program in Fig. 2a-b. The user first defines various units of measure, each represented as a value of the general value-with-units type, QUANT. Next come subtype definitions for each of the distinct dimensionalities of the variables he wishes to declare, with the discriminant constraints giving the dimensionalities. The constants GRAVITY, FRICTION, etc., which are essentially parameters in this problem, can now be expressed in the units just defined. The auxiliary routine GET_QUANT makes it convenient to read in floating point quantities and ascribe to them the appropriate units of measure.

The body of SLIDE illustrates a few interesting points. There are several complex expressions in which quantities with various units of measure are combined. Each addition, subtraction, and assignment checks that the two objects or values involved have the same

* Units are *commensurable* if they have the same dimensionality. I'll use the term *dimensionality* of a unit to refer to the vector of exponents of the *elementary units* that constitute it. For example, if we choose distance to be an elementary unit, then area has units of distance², and hence the dimensionality of area has 2 as the distance component and 0 for all other components. See also Karr and Loveman's paper [3] for a further discussion of the representation of units by vectors of exponents.

```

with UNITS, SOME_IO, SOME_MATH;
use UNITS, SOME_IO, SOME_MATH;
procedure SLIDE is
    -- A simulation of a block sliding down a curved surface

    CM  : constant QUANT := UNIT0;
    FT  : constant QUANT := 30.48*CM;
    SEC : constant QUANT := UNIT1;
    GM  : constant QUANT := UNIT2;

    subtype DISTANCE is QUANT(1,0,0,0);
    subtype TIME      is QUANT(0,1,0,0);
    subtype MASS      is QUANT(0,0,1,0);
    subtype VELOCITY is QUANT(1,-1,0,0);

    GRAVITY      : constant QUANT := 980.7 * CM / SEC**2;
    FRICTION     : constant QUANT := 20.0 * GM / SEC;
    DELTA_X      : constant QUANT := 0.01 * CM;
    DELTA_T      : constant QUANT := 0.1 * SEC;
    BLOCK_MASS   : constant QUANT := 1000.0 * GM;

    X,Y          : DISTANCE := 0.0 * CM;
    V             : VELOCITY := 0.0 * (CM / SEC);
    MAX_TIME     : TIME;

    function HEIGHT(X : DISTANCE) return DISTANCE is
        -- The height of the surface at horizontal coordinate X
        -- (body suppressed)
    end HEIGHT;

    function GET_QUANT(U : QUANT) return QUANT is
        -- The next input value, assumed to have units U.
        Z : FLOAT;
    begin
        GET(Z); return Z*U;
    end GET_QUANT;

```

Figure 2a. Example of a procedure using UNITS: Part 1.

```

begin
  V      := GET_QUANT(CM/SEC);
  MAX_TIME := GET_QUANT(SEC);

  for i in 1 .. INTEGER(+MAX_TIME/DELTA_T) loop declare
    SLOPE      : FLOAT      := +(F(X+DELTA_X)-Y)/DELTA_X;
    COS_ANGLE : FLOAT      := 1.0 / SQRT(1.0 + SLOPE**2);
    NEW_X      : DISTANCE := X + V * DELTA_T * COS_ANGLE;
    NEW_Y      : FLOAT      := F(NEW_X);
    NEW_V      : VELOCITY :=
      V + DELTA_T * (GRAVITY * SLOPE * COS_ANGLE
        - FRICTION * V / BLOCK_MASS);
  begin
    X := NEW_X; V := NEW_V; Y := NEW_Y;
    PUT("At T = " & FLOAT(i)*DELTA_T/SEC & " sec, X = " & X/FT
      & " ft, Y = " & Y/FT & " ft, and V = " & V/(FT/SEC)
      & " ft/sec.");
    NEW_LINE;
  end;
end loop;
end SLIDE;

```

Figure 2b. Example of a procedure using UNITS: Part 2.

constraints (this is not apparent here, but will be when we get to the UNITS package itself.) Likewise, calls to HEIGHT make sure that the argument passed is indeed in units of distance. The upper bound of the **for** loop and the definition of SLOPE illustrate the use of unary '+' as a convenient coercion operator; as you will see, it is overloaded to yield **FLOAT** values from an operand of type **QUANT**, provided that the operand is a scalar (i.e., that its dimensional constraints are all 0.) The PUT statement illustrates another very convenient overloading: binary '/' is overloaded to produce results of type **STRING**, again with a check that the result of division would be a scalar. Hence, the expression Y/FT can be read as "the numeric value of Y in feet." (Also, notice the use of string catenation to effect multiple data arguments to PUT.)

The subtypes chosen reflect only dimensionality, not scaling. For example, we have a subtype for distance, but not one for, say, centimeters. This avoids annoying conversions that would otherwise be necessary for parameter passing or assignment. For example, had we included scaling information in the subtype, it would be difficult to write a function that operated on distances without regard to scaling factor, unless we required that the programmer supply explicit coercions to whatever scale unit that function took, or unless we required that the unconstrained type **UNIT** be the formal parameter's type.

The definitions of **DISTANCE**, **TIME**, **CM**, etc., are clearly independent of the rest of **SLIDE**. For stylistic reasons, therefore, one might want to put these definitions in a separate package. I have chosen not to do so here, since my main concern is with the most general abstraction—unit of measure—apart from any particular choice of a measurement

system.

I think most will agree that a UNITS package supporting this example, especially if it could be implemented efficiently, would go a long way toward providing Ada programmers with true dimensional analysis. Let's consider first the implementation of the functionality displayed above, and then turn to efficiency.

2.2. The UNITS Package

Figure 3 gives a specification for a UNITS package that will support the SLIDE program of the preceding section. Figure 4 gives an implementation for UNITS. Ellipses replace obvious portions.

3. Implementation Considerations

A "naive compiler's" implementation of UNITS would have three obvious problems.

1. Each value of type QUANT would require space for a value of type FLOAT and four values of type INTEGER.
2. The indicated run time checks slow down execution. There is nothing in the Ada Standard, in other words, to guarantee that the UNITS package will perform as we wish.
3. Since the dimensional consistency violations are not illegal in the Ada-technical sense of that word, but rather raise exceptions upon execution, we appear to lose compile time checking.

I will argue that it is perfectly reasonable to expect compilers routinely to give us the desired performance. The key requirements placed on the compiler are first that it pick the proper representation for records with discriminants, and second that it implement constant folding.

3.1. A Space-Efficient Representation of Records with Discriminants

For the UNITS example, the space requirements need not be outrageous. One could assume that large exponents will be rare, and change the UNITS package specification so that QUANT's discriminants take on values in the range -128 to 127. A compiler that packs record fields will then require only four 8-bit bytes for the discriminants. However, even this probably doubles the storage requirements for an individual object, relative to a simple FLOAT value. This becomes most noticeable when dealing with arrays. Let's consider another approach, therefore.

The discriminants of a constrained record object are fixed, and are identical for record objects having the same declared subtype. This suggests that the compiler can implement the type QUANT, for example, as *two* types—one containing the discriminant fields, and the other containing the single field V. For purposes of exposition, let me use italics to indicate compiler-generated names, and call these two types QUANT_*discrim.type* and QUANT_*value.type*:

```
type QUANT_discrim.type is
  record D0,D1,D2,D3 : INTEGER; end record;
type QUANT_value.type is
  record V : FLOAT; end record;
```

package UNITS is

type QUANT(D0,D1,D2,D3 : INTEGER) is
record V : FLOAT; end record;

-- Convenient abbreviations.

subtype SCALAR is QUANT(0,0,0,0);
UNIT0 : constant QUANT := (1,0,0,0,1.0);
UNIT1 : constant QUANT := (0,1,0,0,1.0);
UNIT2 : constant QUANT := (0,0,1,0,1.0);
UNIT3 : constant QUANT := (0,0,0,1,1.0);

-- Standard arithmetic

function "+"(L,R : QUANT) return QUANT;
function "-"(L,R : QUANT) return QUANT;
function "-"(L : QUANT) return QUANT;
function "*" (L : FLOAT; R : QUANT) return QUANT;
function "*" (L,R : QUANT) return QUANT;
function "/"(L : FLOAT; R : QUANT) return QUANT;
function "/"(L : QUANT; R : FLOAT) return QUANT;
function "/"(L,R : QUANT) return QUANT;
function "*"(L : QUANT; R : INTEGER) return QUANT;**

-- Coercions

function "/"(L,R : QUANT) return STRING;
function "+"(L : SCALAR) return FLOAT;
function "-"(L : SCALAR) return FLOAT;

-- Relationals

function "<"(L,R : QUANT) return BOOLEAN;
...

-- Exceptions

-- Relationals, binary "+" and "-", and "/" yielding STRING raise
-- CONSTRAINT_ERROR if L and R have differing subtypes.

pragma INLINE("+","-","/","*","","<","..");**
end UNITS;

Figure 3. Specification for the UNITS Package.

```

with TEXT_IO;
package body UNITS is

    package UNITS_FLOAT is new TEXT_IO.FLOAT_IO(FLOAT);

    function "+"(L,R : QUANT) return QUANT is
    begin
        if L.D0 /= R.D0 or L.D1 /= R.D1 or
           L.D2 /= R.D2 or L.D3 /= R.D3
           then raise CONSTRAINT_ERROR;
        end if;
        return (L.D0,L.D1,L.D2,L.D3,L.V+R.V);
    end;

    function "*" (L,R : QUANT) return QUANT is
    begin
        return (L.D0+R.D0,L.D1+R.D1,L.D2+R.D2,L.D3+R.D3,
                L.V*R.V);
    end;

    function "*" (L : FLOAT; R : QUANT) return QUANT is
    begin
        return (R.D0,R.D1,R.D2,R.D3,L*R.V);
    end;

    function "**" (L : FLOAT; R : INTEGER) return QUANT is
    begin
        return (L.D0*R, L.D1*R, L.D2*R, L.D3*R, L.V**R);
    end;

    function "/"(L,R : QUANT) return STRING;
    use UNITS_FLOAT;
    FLOAT_WIDTH : constant FIELD :=
        DEFAULT_FORE+DEFAULT_AFT+DEFAULT_EXP+2;
    Z : constant SCALAR := L/R;
    STRING_VALUE : STRING(1 .. FLOAT_WIDTH);
    begin
        PUT(TO => STRING_VALUE, ITEM => Z.V);
        return STRING_VALUE;
    end;
    ...
end UNITS;

```

Figure 4. Body of the UNITS package.

The compiler represents each subtype indication of **QUANT** with a *descriptor*—a value of a type denoted **QUANT_discrim_type**. It represents each object of type **QUANT** with an object of type **QUANT_value_type**.

When several objects of type **QUANT** use the same subtype indication, only one descriptor of type **QUANT_discrim_type** is needed. For example, consider the following declarations.

```
subtype DISTANCE is QUANT(1,0,0,0);  
type DISTANCE_VECT is array (INTEGER range <>) of DISTANCE;  
B : DISTANCE_VECT(1 .. N);
```

The compiler allocates space as follows.

```
DISTANCE_subtype : constant QUANT_discrim_type := (1,0,0,0);  
type DISTANCE_VECT is array (INTEGER range <>) of QUANT_value_type;  
B : DISTANCE_VECT(1 .. N);
```

The reason this works—why one does not need to store the discriminants with each element—is that the compiler always knows where to go to find the subtype of an element of **B**. In those places where it needs this subtype information, such as in doing constraint checking, it looks at **DISTANCE_subtype**. If **B** is passed as an argument to a subprogram, the type **DISTANCE_VECT** must necessarily be known to the subprogram, and hence **DISTANCE_subtype** will still be available.

When a formal parameter of a subprogram has a constrained subtype of **QUANT** (as in the **HEIGHT** function in Fig. 2a) only the **QUANT_value_type** portion of the actual parameter need be passed. When the subtype of the formal is unconstrained, both the **QUANT_discrim_type** and **QUANT_value_type** portions must be passed.

This latter fact has actually misled some compiler implementors into declaring the implementation proposed here a “pessimization.” The grounds for their contention appear to be that the compiler must pass two addresses instead of one for an unconstrained formal (assuming pass-by-reference.) However, the cost of passing a single constant address is quite small—easily swamped by the general call overhead and the cost of executing the subprogram body itself. For example, using the C compiler on a VAX/750, I have compared two simple programs, one of which calls a null function with two address parameters and one of which calls a null function with one address parameter. There was a 10% difference in execution time. Even assuming that 10% of all execution time in an Ada program is the call-return overhead for (non-inlined) subprogram calls with unconstrained record parameters (a very generous estimate), this translates to a 1% penalty in execution time. This estimate is a rather conservative upper bound, since in practice the bodies of subprograms contain something. Furthermore, in the case of inline subprograms, of course, there needn't be any increase in cost occasioned by separating the discriminants.

There are various complications that arise when we consider record fields of type **QUANT** that are constrained by discriminants, allocators for type **QUANT**, or unconstrained objects (which is not actually an issue for **QUANT**, but is in the general case for this representation.) However, these do not change the basic model fundamentally. In the case of a record field of type **QUANT** that is constrained by a discriminant, the subtype becomes part of the descriptor of the enclosing record. Allocated objects and unconstrained objects are both represented by records containing both a descriptor and a

value part. The report by Zorn [5] contains a more detailed description of a representation like this. Finally, to be absolutely complete, one would probably want to introduce implementation-dependent attributes, such as `QUANT'DEScriptor.SIZE` and `X'DEScriptor.ADDRESS` for implementing such things as basic I/O routines for arbitrary types of objects.

Considering the space-efficiency of the implementation described here, it appears an entirely reasonable one for compilers to use. When they do so, constant folding will eliminate almost all run time checks.

3.2. Constant Folding

The term *constant folding* means replacement of expressions whose values are known at compile time by their values. This includes such things as replacing `INTEGER'LAST-1` by `2147483646` and replacing `UNIT0.D1` by `0`. It also includes expressions with "control flow" values, as in the replacement of

```
if DEBUGGING then ASSERT(X > Y); end if;
```

by the null statement when `DEBUGGING` is defined to be the static constant `FALSE`.

The application of constant folding to uses of the `UNITS` package should be fairly clear. First, we are usually dealing with objects that have static subtypes (such as `DISTANCE` or `MASS`.) For example, consider the implicit constraint checking that occurs upon the assignment

```
X := NEW_X;
```

from Fig. 2b. After looking up the subtypes of `X` and `NEW_X`, the compiler might expand this assignment to

```
if DISTANCE_subtype = DISTANCE_subtype
then X := NEW_X;
else raise CONSTRAINT_ERROR;
end if;
```

After constant folding this collapses to just

```
X := NEW_X;
```

Furthermore, the `INLINE` pragma asks the compiler to expand the operators on `QUANTs` in line. Hence, an assignment such as

```
X := X + DELTA_X;
```

ultimately might expand into the code shown in Fig. 5. For authenticity, I've made this a general form of inline expansion. The compiler introduces the quantities with names beginning `X0`, `Y0`, and `RETURN`. We are assuming the representation introduced in the last section, so that `X` and `DELTA_X` are represented by values of type `QUANT_value.type`.

Using constant folding, the whole thing collapses to something close to what the programmer wanted to have happen.

```
RETURN_value := (V => X0.V + Y0.V);
X := RETURN_value;
```

(This circumlocution is an artifact of the general expansion I chose and has nothing to do with the handling of discriminants. It will collapse to a single assignment if the compiler

```

declare
  X0_discrim      : QUANT_discrim_type renames CM_subtype;
  X0_value        : QUANT_value_type  renames X;
  Y0_discrim      : QUANT_discrim_type renames CM_subtype;
  Y0_value        : QUANT_value_type  renames DELTA_X;
  RETURN_discrim  : QUANT_discrim_type;
  RETURN_value    : QUANT_value_type;
begin
  if X0_discrim /= Y0_discrim
    then raise CONSTRAINT_ERROR;
  end if;
  RETURN_discrim :=
    (X0_discrim.D0, X0_discrim.D1, X0_discrim.D2, X0_discrim.D3);
  RETURN_value  := (V => X0_value.V + Y0_value.V);
  if X_discrim = RETURN_discrim
    then X := RETURN_value;
  else raise CONSTRAINT_ERROR;
  end if;
end;

```

Figure 5. Possible compiler-expanded version of $X := X + \text{DELTA_X}$.

applies a little value propagation, peephole optimization, or a better expansion of the inline function.)

Constant folding, as it is used here, doesn't require any fancy global flow analysis. However, it does differ from common forms of constant folding in that it involves composite objects rather than just scalar values. Folding of scalar expressions is, in fact, required by the Ada language itself, because of the rules governing static expressions. For the optimizations discussed in this section, extending constant folding to composite objects requires little more machinery. The only additional objects that need to be handled are those with types such as *QUANT_discrim_type*—i.e., record objects without discriminants, all of whose fields have discrete types and static values.

If a compiler writer is willing to fold more ambitiously, it is possible to get some of the improvements mentioned here without using the separated discriminant representation of the preceding section. This requires tracking static values of individual fields in a record object. Using such techniques, and taking advantage of the observation that the values of constrained discriminants can't change, the addition

$X := X + \text{DELTA_X};$

may be reduced eventually to the equivalent of

$X.V := X.V + \text{DELTA_X}.V;$

Of course, one still incurs a space penalty, as well as an execution-time penalty for initializing all discriminant fields of variables. However, as you can see, the execution time required for many operations will still be about the same as for the predefined operations on *FLOAT*.

3.3. Reporting Errors

I still have not answered one question: although we seem to handle the *absence* of errors at compilation time, what happens if there are errors—how will they be reported? Constant folding will reduce a dimensionally inconsistent statement, such as

```
X := 3.0 * CM / SEC;
```

to a simple **raise CONSTRAINT_ERROR**. Technically, however, the resulting program is completely legal and can be executed without any problem (assuming that it is written to remain unfazed upon encountering **CONSTRAINT_ERROR**.)

Here again we must consider what it is reasonable for a compiler to do. The design of Ada has a heavy bias towards considering the raising of an exception to be a kind of error indication. Hence, it is reasonable to assume that statements and declarations that can be determined to raise exceptions ought to be brought to the programmer's attention as soon as possible. More precisely, if there are control paths through a subprogram that do not raise exceptions, but all of these paths are eliminated by constant folding, then the compiler should produce a warning to this effect.

Of course, even if compilers fail to provide such warnings, the result is still that exceptions will occur during execution. The error will be discovered, although not at the most convenient time.

4. Various Improvements and Weaknesses

For the sake of cleanliness, I made the **UNITS** package fairly spare. There are a number of enhancements that might be useful.

The package now uses the pre-defined type **FLOAT** for all arithmetic. For generality, it might be advisable to make **UNITS** generic in the floating point type to be used. It would be very difficult, unfortunately (or fortunately, depending on your political bias) to extend the package to general fixed point types. The problem is that in fixed point arithmetic, multiplication and division operators get introduced automatically, something that is not possible to duplicate in a user-defined package.

As it stands, **UNITS** handles only integral dimensionalities. To extend to arbitrary rational dimensions would require doubling the number of discriminants. For best results, it would also require a constant-folding scheme that could handle a g.c.d. computation, which is probably too much to ask. Programmers who need units of square root of distance, for example, can get the effect by defining

```
subtype DISTANCE is QUANT(2,0,0,0);
```

If fractional dimensions are common, it might be desirable to introduce a **SQRT** function into the package (which would accept only even dimensionalities.)

The definition of subtypes as explicit combinations of discriminants is a bit awkward. Unfortunately, I haven't found a good alternative. The best seems to be to introduce an embedded generic package in **UNITS**, as follows.

```

generic
  T : in QUANT;
package A_UNIT is
  subtype UNIT is QUANT(T.D0,T.D1,T.D2,T.D3);
end;

```

This allows the user to write subtypes such as *DISTANCE* or *VELOCITY* more symbolically, relating them to each other and to *UNIT0*, *UNIT1*, etc.

```

package DIST is new A_UNIT(CM);
subtype DISTANCE is DIST.UNIT;
package VELO is new A_UNIT(CM/SEC);
subtype VELOCITY is VELO.UNIT;

```

Unfortunately, this is rather wordy, and does not improve the clarity of the program to speak of.

5. Conclusions

I have known about this example for a number of years. What finally prompted me to publish it here was an article by R. A. O'Keefe [4] in which he remarks parenthetically,

I am astounded that this longstanding idea [physical dimensions as types] did not make it into ADA [sic], so much else of less potential benefit having been included.

In view of the package presented here, the quotation above is one more illustration of how easy it is to underestimate the power of abstraction mechanisms. Dimensions as types have no place in Ada because they are *definable* in Ada without any extra language features.

The particular abstraction mechanisms of Ada that made this definition possible are operator definition, overloading, and type parameterization (the Ada term being "discriminant constraints.") Operator definition and overloading made possible the use of familiar arithmetic notation and allowed concise notation for type coercions. Type parameterization made it possible to supply extra information about the value domain of a given variable at declaration time and provided automatic consistency checks in the form of constraint testing upon assignment and parameter passing.

A good deal hinges on the behavior of our compilers. The definition will *work* in any case, but we are unlikely to be completely satisfied unless it works as well as we know is possible. The required representation is probably a good idea quite apart from this use of it. Its space requirements are better than the "naive" representation, and its execution time requirements are probably indistinguishable in practice. I have argued that the features expected of the compiler are perfectly reasonable. Some mechanism for constant folding is essentially mandated by the Ada Standard for other purposes, and its extension to descriptors is not difficult. Compilation-time warnings about inevitable exceptions are also things we might expect of good compilers, given the Ada goal of detecting as many errors as possible statically.

Of course, however reasonable these features may be, they won't be universal. I hope, though, that the prospect of cost-free dimensional analysis will prompt users to agitate for compilers that have them.

Acknowledgements

I would like to thank Ron Brender of Digital Equipment Corporation, John Goode-nough of SofTech, Inc., and Ben Zorn of the University of California for their comments on drafts of this paper.

References

- [1] Narain Gehani, "Ada's Derived Types and Units of Measure." *Software—Practice & Experience*, 15, 6 (June 1985), pp. 555-569.
- [2] Paul N. Hilfinger, *Abstraction Mechanisms and Language Design*. The MIT Press, 1983.
- [3] Michael Karr and David B. Loveman III, "Incorporation of Units into Programming Languages." *Comm. ACM* 21,5 (May, 1978), pp. 385-391.
- [4] R. A. O'Keefe, "Alternatives to Keyword Parameters." *SIGPLAN Notices*, 20, 6 (June 1985), pp. 26-32.
- [5] Benjamin G. Zorn, *Experience with Ada Code Generation*, Technical Report UCB/CSD 85/249, Computer Science Division, University of California, Berkeley, June, 1985.